developerWorks®

Migrate Win32 C/C++ applications to Linux on POWER, Part 3: Semaphores

Nam Keung

March 31, 2005

Migrate your Win32 C/C++ applications to Linux™ on POWER™ and learn how to map Win32 to Linux with respect to semaphore application program interfaces (APIs). Follow along as Nam Keung walks you through detailed code examples outlining this process.

Introduction

This article, third in a series, is about migrating the Win32 C/C++ application to Linux on POWER with respect to semaphore APIs. Part 1 of this series addressed Win32 API mapping and Part 2 focused on how to map Win32 to Linux with respect to mutex APIs. You are encouraged to read Part 1 and Part 2 of this series before proceeding further.

Semaphores

A semaphore is a resource that contains an integer value. Semaphores allow synchronization of processes by testing and setting the integer value in a single atomic operation. Usually, the main use of a semaphore is to synchronize a thread?s action with other threads. This is also a useful technique for coordinating or synchronizing activities in which multiple processes compete for the same operating system resources.

Linux provides Portable Operating System Interface (POSIX) semaphores, as well as pthread conditional variables to map the Win32 semaphore APIs. Both have their share of pros and cons. It is up to your discretion to use either one based on application logic. The various points to consider in the mapping process of the event semaphore are:

- *Type of semaphore:* Win32 supports both named and un-named event semaphores. The named semaphores are shared across processes. Linux does not support this option. An Inter-Process Communication (IPC) message queues sample code listed in this article to show you how to work around it.
- *Initial state:* In Win32, the semaphore might have an initial value. In Linux, the POSIX semaphore supports this functionality, but the pthreads do not. You need to consider this when using pthreads.
- *Timeout:* Win32 event semaphores support timed wait. In Linux, the POSIX semaphore implementation only supports indefinite wait (blocking). The pthreads implementation supports

both blocking and timeouts. The `pthread_cond_timedwait()` call provides a timeout value during wait, and the `pthread_cond_wait()` is used for indefinite wait.
- *Signaling:* In Win32, signaling a semaphore wakes up all the threads that are waiting on the semaphore. In Linux, the POSIX thread implementation wakes up only one thread at a time. The pthreads implementation has a `pthread_cond_signal()` call that wakes up one thread and a `pthread_cond_broadcast()` call that signals all the threads waiting on the semaphore.

Table 1. Semaphore mapping table

Win32	pthread Linux	POSIX

`CreateSemaphore`	`pthread_mutex_init(&(token)->mutex, NULL))`
	`pthread_cond_init(&(token)->condition, NULL))`
		`sem_init`

`CloseHandle (semHandle)`	`pthread_mutex_destroy(&(token->mutex))`
	`pthread_cond_destroy(&(token->condition))`
		`sem_destroy`

`ReleaseSemaphore(semHandle, 1, NULL)`	`pthread_cond_signal(&(token->condition))`	`sem_post`

`WaitForSingleObject(semHandle, INFINITE)`
`WaitForSingleObject(semHandle, timelimit)`
	`pthread_cond_wait(&(token->condition), &(token->mutex))`
	`pthread_cond_timedwait(&(token ->condition), &(token->mutex))`
		`sem_wait`
		`sem_trywait`

Condition variable

A condition variable enables developers to implement a condition in which a thread executes and then blocked. The Microsoft® Win32 interface does not support condition variables natively. To work around this omission, I use the POSIX condition variable emulations synchronization primitives, which are outlined in the series of articles. In Linux, it guarantees the threads blocked on the condition will be unblocked when the condition changes. It also allows you to unlock the mutex and wait on the condition variable atomically, without the possible intervention of another thread. However, a mutex should accompany each condition variable. Table 1 above displays the pthread condition variable for synchronization between threads.

Creating a semaphore

In Win32, the `CreateSemaphore` function creates a named or unnamed semaphore object. Linux does not support named semaphores.

Listing 1. Creating a semaphore

```
HANDLE CreateSemaphore (
 LPSECURITY_ATTRIBUTES lpSemaphoreAttributes,
 LONG   lInitialCount,
 LONG   lMaximunCount,
 LPCTSTR   lpName
);
```

In Linux, the call `sem_init()` also creates a POSIX semaphore:

Listing 2. POSIX semaphore

```
int sem_init(sem_t *sem, int pshared, unsigned int value
```

Linux uses the `pthread_condition_init` call to create a semaphore object within the current process that maintains a count between zero and a maximum value. The count is decremented each time a thread completes a wait for the semaphore object and incremented each time a thread releases the semaphore. When the count reaches zero, the state of the semaphore object becomes non-signaled.

Listing 3. `pthread_condition_init` call to create a semaphore object

```
int pthread_cond_init(pthread_cond_t *cond, const pthread_condattr_t *attr);
```

Listing 4. Win32 sample code

```
HANDLE semHandle;

semHandle = CreateSemaphore(NULL, 0, 256000, NULL);
     /* Default security descriptor     */
if( semHandle == (HANDLE) NULL)
     /* Semaphore object without a name */
{
     return RC_OBJECT_NOT_CREATED;
}
```

Listing 5. Equivalent Linux code

```
typedef struct
{
 pthread_mutex_t mutex;
 pthread_cond_t  condition;
 int   semCount;
}sem_private_struct, *sem_private;

sem_private    token;

token = (sem_private) malloc(sizeof(sem_private_struct));

if(rc = pthread_mutex_init(&(token->mutex), NULL))
{
 free(token);
 return RC_OBJECT_NOT_CREATED;
}

if(rc = pthread_cond_init(&(token->condition), NULL))
{
 pthread_mutex_destroy( &(token->mutex) );
 free(token);
 return RC_OBJECT_NOT_CREATED;
}

token->semCount = 0;
```

Destroying an event semaphore

Win32 uses `CloseHandle` to delete the semaphore object created by the `CreateSemaphore`.

Listing 6. Destroying an event semaphore

```
BOOL CloseHandle (HANDLE hObject);
```

Linux POSIX semaphores use `sem_destroy()` to destroy the unnamed semaphore.

Listing 7. `sem_destroy()`

```
int sem_destroy(sem_t *sem);
```

In Linux pthreads, the `pthread_cond_destroy()` is used to destroy the conditional variable.

Listing 8. `pthread_cond_destroy()`

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

Listing 9. Win32 code and euivalent Linux code

Win32 code	Equivalent Linux code

`CloseHandle(semHandle);`	`pthread_mutex_destroy(&(token->mutex));`
	`pthread_cond_destroy(&(token->condition));`
	`free (token);`

Posting an event semaphore

In Win32, the `ReleaseSemaphore` function increases the count of the specified semaphore object by a specified amount.

Listing 10. `ReleaseSemaphore` function

```
BOOL ReleaseSemaphore(
 HANDLE hSemaphore,
 LONG  lReleaseCount,
 LPLONG lpPreviousCount
);
```

Linux POSIX semaphores use `sem_post()` to post an event semaphore. This wakes up any of the threads blocked on the semaphore.

Listing 11. `sem_post()`

```
int sem_post(sem_t * sem);
```

In Linux, `pthread_cond_signal` wakes up a thread waiting on a conditional variable. Linux calls this function to post one event completion for the semaphore identified by the object. The calling thread increments the semaphore. If the semaphore value is incremented from zero and there is any threads blocked in the `pthread_cond`, wait for the semaphore because one of them is awakened. By default, the implementation can choose any of the waiting threads.

Listing 12. `pthread_cond_signal`

```
int pthread_cond_signal(pthread_cond_t *cond);
```

Listing 13. Win32 code and equivalent Linux code

Win32 code	Equivalent Linux code

`ReleaseSemaphore(semHandle, 1, NULL)`	`if (rc = pthread_mutex_lock(&(token->mutex)))`
	`return RC_SEM_POST_ERROR;`
	`token->semCount ++;`
	`if (rc = pthread_mutex_unlock(&(token->mutex)))`
	`return RC_SEM_POST_ERROR;`
	`if (rc = pthread_cond_signal(&(token->condition)))`
	`return RC_SEM_POST_ERROR;`

Waiting an event semaphore

Win32 calls the `WaitForSingleObject` function to wait for an event completion on the indicated semaphore. You can use this method when waiting on a single thread synchronization object. The method is signaled when the object is set to signal or the time out interval is finished. If the time interval is INFINITE, it waits infinitely.

Listing 14. `WaitForSingleObject` function

```
DWORD WaitForSingleObject(
 HANDLE hHANDLE,
 DWORD dwMilliseconds
);
```

Use the `WaitForMultipleObjects` function to wait for multiple objects signaled. In the Semaphore thread synchronization object, the object is non-signaled when the counters go to zero.

Listing 15. `WaitForMultipleObjects` function

```
DWORD WaitForMultipleObjects(
 DWORD nCount,
 Const HANDLE* lpHandles,
 BOOL bWaitAll,
 DWORD dwMilliseconds
);
```

Linux POSIX semaphores use `sem_wait()` to suspend the calling thread until the semaphore has a non-zero count. Then it atomically decreases the semaphore count.

Listing 16. `sem_wait()` function

```
int sem_wait(sem_t * sem);
```

The timeout option is not available in the POSIX semaphore. However, you can achieve this by issuing a non-blocking `sem_trywait()` within a loop, which counts the timeout value.

Listing 17. `sem_trywait()` function

```
int sem_trywait(sem_t  * sem);
```

In Linux, the `pthread_cond_wait()` blocks the calling thread. The calling thread decrements the semaphore. If the semaphore is zero when the `pthread_cond_wait` is called, the `pthread_cond_wait()` blocks until another thread increments the semaphore.

Listing 18. `pthread_cond_wait()` function

```
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t  *mutex);
```

The `pthread_cond_wait` function first releases the associated `external_mutex of type pthread_mutex_t`, which must be held when the caller checks the condition expression.

Listing 19. Win32 code and equivalent Linux code

Win32 code	Equivalent Linux code

DWORD retVal;
retVal = WaitForSingleObject(semHandle, INFINITE);
if (retVal == WAIT_FAILED) return RC_SEM_WAIT_ERROR
	if (rc = pthread_mutex_lock(&(token->mutex)))
	return RC_SEM_WAIT_ERROR;
	while (token->semCount <= 0)
	{
	rc = pthread_cond_wait(&(token->condition), &(token->mutex));
	if (rc &&errno != EINTR)
	break;
	}
	token->semCount--;
	if (rc = pthread_mutex_unlock(&(token->mutex)))
	return RC_SEM_WAIT_ERROR;

If you need to block the calling thread for a specific time, then use the `pthread_cond_timewait` to block the thread. This method is called to wait for an event completion on the indicated semaphore, with a specified time.

Listing 20. pthread_cond_timewait

```
 int pthread_cond_timewait(
 pthread_cond_t  *cond,
 pthread_mutex_t  *mutex,
 timespec  *tm
);
```

Listing 21. Win32 code and equivalent Linux code

Win32 code	Equivalent Linux code

retVal = WaitForSingleObject(SemHandle, timelimit);
if (retVal == WAIT_FAILED)
return RC_SEM_WAIT_ERROR;
if (retVal == WAIT_TIMEOUT)
return RC_TIMEOUT;
	int rc;
	struct timespec tm;
	struct timeb tp;
	long sec, millisec;
	if (rc = pthread_mutex_lock(&(token->mutex)))
	return RC_SEM_WAIT_ERROR;
	sec = timelimit / 1000;
	millisec = timelimit % 1000;
	ftime(&tp);
	tp.time += sec;
	tp.millitm += millisec;

```
if( tp.millitm > 999 )
{
tp.millitm -= 1000;
tp.time++;
}
tm.tv_sec = tp.time;
tm.tv_nsec = tp.millitm * 1000000 ;
while (token->semCount <= 0)
{
rc = pthread_cond_timedwait(&(token->condition),
&(token->mutex), &tm);
if (rc && (errno != EINTR) )
break;
}
if ( rc )
{
if ( pthread_mutex_unlock(&(token->mutex)) )
return RC_SEM_WAIT_ERROR );
if ( rc == ETIMEDOUT) /* we have a time out */
return RC_TIMEOUT );
return RC_SEM_WAIT_ERROR );
}
token->semCount--;
if (rc = pthread_mutex_unlock(&(token->mutex)))
return RC_SEM_WAIT_ERROR;
```

POSIX semaphore sample code

Listing 22 uses POSIX semaphores to implement synchronization between threads A and B:

Listing 22. POSIX semaphore sample code

```
sem_t sem; /* semaphore object */
int irc;   /* return code */

/* Initialize the semaphore - count is set to 1*/
irc = sem_init (sem, 0,1)

...

/* In Thread A */

/* Wait for event to be posted */

sem_wait (&sem);

/* Unblocks immediately as semaphore initial count was set to 1 */

 .......

/* Wait again for event to be posted */

sem_wait (&sem);

/* Blocks till event is posted */

/* In Thread B */
/* Post the semaphore */
...

irc = sem_post (&sem);

/* Destroy the semaphore */
irc = sem_destroy(&sem);
```

Intra-process semaphore sample code

Listing 23. Win32 intra-process semaphore sample code

```
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

void thrdproc      (void   *data);  // the thread procedure (function)
     to be executed

HANDLE   semHandle;

int
main( int argc, char **argv )
{
        HANDLE    *threadId1;
        HANDLE    *threadId2;
        int        hThrd;
        unsigned   stacksize;
        int     arg1;

 if( argc < 2 )
  arg1 = 7;
 else
  arg1 = atoi( argv[1] );

 printf( "Intra Process Semaphor test.\n" );
 printf( "Start.\n" );

 semHandle = CreateSemaphore(NULL, 1, 65536, NULL);
        if( semHandle == (HANDLE) NULL)
 {
  printf("CreateSemaphore error: %d\n", GetLastError());
 }

 printf( "Semaphor created.\n" );

        if( stacksize < 8192 )
            stacksize = 8192;
        else
            stacksize = (stacksize/4096+1)*4096;

        hThrd = _beginthread( thrdproc, // Definition of a thread entry
                                  NULL,
                             stacksize,
                            "Thread 1");

        if (hThrd == -1)
            return RC_THREAD_NOT_CREATED);

        *threadId1 = (HANDLE) hThrd;

         hThrd = _beginthread( thrdproc, // Definition of a thread entry
                                   NULL,
                              stacksize,
                              ?Thread 2");

         if (hThrd == -1)
            return RC_THREAD_NOT_CREATED);

         *threadId2 = (HANDLE) hThrd;

  printf( "Main thread sleeps 5 sec.\n" );

         sleep(5);
         if( ! ReleaseSemaphore(semHandle, 1, NULL) )
  printf("ReleaseSemaphore error: %d\n", GetLastError());

         printf( "Semaphor released.\n" );
         printf( "Main thread sleeps %d sec.\n", arg1 );

         sleep (arg1);

       if( ! ReleaseSemaphore(semHandle, 1, NULL) )
  printf("ReleaseSemaphore error: %d\n", GetLastError());

 printf( "Semaphor released.\n" );
 printf( "Main thread sleeps %d sec.\n", arg1 );
 sleep (arg1);

        CloseHandle(semHandle);

 printf( "Semaphor deleted.\n" );
 printf( "Main thread sleeps 5 sec.\n" );

        sleep (5);
        printf( "Stop.\n" );

 return OK;
}

void
thread_proc( void *pParam )
{

 DWORD  retVal;

 printf( "\t%s created.\n", pParam );

        retVal = WaitForSingleObject(semHandle, INFINITE);

        if (retVal == WAIT_FAILED)
               return RC_SEM_WAIT_ERROR;

 printf( "\tSemaphor blocked by %s. (%lx)\n", pParam, retVal);
 printf( "\t%s sleeps for 5 sec.\n", pParam );
 sleep(5);

 if( ! ReleaseSemaphore(semHandle, 1, NULL) )
                printf("ReleaseSemaphore error: %d\n", GetLastError());

 printf( "\tSemaphor released by %s.)\n", pParam);
}
```

Listing 24. Equivalent Linux intra-process semaphore sample code

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <pthread.h>
#include <errno.h>

void  thread_proc (void * data);

pthread_mutexattr_t     attr;
pthread_mutex_t         mutex;

typedef struct
{
        pthread_mutex_t         mutex;
        pthread_cond_t          condition;
        int                     semCount;
}sem_private_struct, *sem_private;

sem_private     token;

int main( int argc, char **argv )
{
        pthread_t             threadId1;
        pthread_t             threadId2;
        pthread_attr_t        pthread_attr;
        pthread_attr_t        pthread_attr2;

        int   arg1;
        int                   rc;

 if( argc < 2 )
  arg1 = 7;
 else
  arg1 = atoi( argv[1] );

      printf( "Intra Process Semaphor test.\n" );
      printf( "Start.\n" );

      token =(sem_private)  malloc (sizeof (sem_private_struct));

      if(rc = pthread_mutex_init( &(token->mutex), NULL))
      {
                free(token);
                return 1;
      }

      if(rc = pthread_cond_init(&(token->condition), NULL))
      {
           printf( "pthread_condition ERROR.\n" );
           pthread_mutex_destroy( &(token->mutex) );
           free(token);
           return 1;
      }

      token->semCount = 0;

      printf( "Semaphor created.\n" );

      if (rc = pthread_attr_init(&pthread_attr))
      {
           printf( "pthread_attr_init ERROR.\n" );
           exit;
      }

      if (rc = pthread_attr_setstacksize(&pthread_attr, 120*1024))
      {
           printf( "pthread_attr_setstacksize ERROR.\n" );
           exit;
      }

      if (rc = pthread_create(&threadId1,
                           &pthread_attr,
            (void*(*)(void*))thread_proc,
                               "Thread 1" ))
      {
           printf( "pthread_create ERROR.\n" );
           exit;
      }

      if (rc = pthread_attr_init(&pthread_attr2))
      {
           printf( "pthread_attr_init2 ERROR.\n" );
```

```
            exit;
        }

        if (rc = pthread_attr_setstacksize(&pthread_attr2, 120*1024))
        {
            printf( "pthread_attr_setstacksize2 ERROR.\n" );
            exit;
        }

        if (rc = pthread_create(&threadId2,
                              &pthread_attr2,
             (void*(*)(void*))thread_proc,
                                   "Thread 2" ))
        {
            printf( "pthread_CREATE ERROR2.\n" );
            exit ;  // EINVAL, ENOMEM
        }

        printf( "Main thread sleeps 5 sec.\n" );
        sleep( 5 );

        if (rc =  pthread_mutex_lock(&(token->mutex)))
        {
            printf( "pthread_mutex_lock ERROR 1.\n" );
            return 1;
        }

        token->semCount ++;

        if (rc = pthread_mutex_unlock&(token->mutex)))
        {
            printf( "pthread_mutex_unlock ERROR 1.\n" );
            return 1;
        }

        if (rc = pthread_cond_signal(&(token->condition)))
        {
            printf( "pthread_cond_signal ERROR1.\n" );
            return 1;
        }

        printf( "Semaphor released.\n" );
        printf( "Main thread sleeps %d sec.\n", arg1 );
        sleep( arg1 );

        if (rc =  pthread_mutex_lock(&(token->mutex)))
        {
            printf( "pthread_mutex_lock ERROR.\n" );
            return 1;
        }

        token->semCount ++;

        if (rc = pthread_mutex_unlock(&(token->mutex)))
        {
            printf( "pthread_mutex_lock ERROR.\n" );
            return 1;
        }

        if (rc = pthread_cond_signal(&(token->condition)))
        {
            printf( "pthread_cond_signal ERROR.\n" );
            return 1;
        }

        printf( "Semaphor released.\n" );
        printf( "Main thread sleeps %d sec.\n", arg1 );
```

```
      sleep( arg1 );

      pthread_mutex_destroy(&(token->mutex));
      pthread_cond_destroy(&(token->condition));

      printf( "Semaphor deleted.\n" );
      printf( "Main thread sleeps 5 sec.\n" );
      sleep( 5 );
      printf( "Stop.\n" );

      return 0;
}

void
thread_proc( void *pParam )
{
 int rc;

      printf( "\t%s created.\n", pParam );

      if (token == (sem_private) NULL)
          return ;

      if (rc =  pthread_mutex_lock(&(token->mutex)))
      {
           printf( "pthread_mutex_lock ERROR2.\n" );
           return ;
      }

      while (token->semCount <= 0)
      {
                rc = pthread_cond_wait(&(token->condition), &(token->mutex));
                if (rc && errno != EINTR )
                        break;
      }
      if( rc )
      {
                pthread_mutex_unlock(&(token->mutex));
                printf( "pthread_mutex_unlock ERROR3.\n" );
                return;
      }
      token->semCount--;

      if (rc = pthread_mutex_unlock(&(token->mutex)))
      {
           printf( "pthread_mutex_lock ERROR.\n" );
           return ;
      }

      printf( "\tSemaphor blocked by %s. (%lx)\n", pParam, rc );
      printf( "\t%s sleeps for 5 sec.\n", pParam );
      sleep( 5 );

      if (rc =  pthread_mutex_lock(&(token->mutex)))
      {
           printf( "pthread_mutex_lock ERROR.\n" );
           return ;
      }

      token->semCount ++;

      if (rc = pthread_mutex_unlock(&(token->mutex)))
      {
           printf( "pthread_mutex_unlock ERROR.\n" );
           return ;
      }
      if (rc = pthread_cond_signal(&(token->condition)))
      {
           printf( "pthread_cond_signal ERROR.\n" );
           return ;
      }

      printf( "\tSemaphor released by %s. (%lx)\n", pParam, rc );
```

Inter-process semaphore sample code

Listing 25. Win32 Inter-process semaphore process 1 sample code

```
#include <stdio.h>
#include <windows.h>

#define WAIT_FOR_ENTER  printf( "Press ENTER\n" );getchar()

int main()
{
       HANDLE  semaphore;
       int  nRet;
       DWORD          retVal;

       SECURITY_ATTRIBUTES  sec_attr;

 printf( "Inter Process Semaphore test - Process 1.\n" );
 printf( "Start.\n" );

 sec_attr.nLength              = sizeof( SECURITY_ATTRIBUTES );
 sec_attr.lpSecurityDescriptor = NULL;
 sec_attr.bInheritHandle       = TRUE;

 semaphore = CreateSemaphore( &sec_attr, 1, 65536, ?456789" );
 if( semaphore == (HANDLE) NULL )
  return RC_OBJECT_NOT_CREATED;

       printf( "Semaphore created. (%lx)\n", nRet );

 WAIT_FOR_ENTER;

 if( ! ReleaseSemaphore(semaphore, 1, NULL) )
  return SEM_POST_ERROR;

       printf( "Semaphore Posted. \n");

 WAIT_FOR_ENTER;

 retVal = WaitForSingleObject (semaphore, INFINITE );
        if (retVal == WAIT_FAILED)
              return SEM_WAIT_ERROR;

 printf( "Wait for Semaphore. \n");

       WAIT_FOR_ENTER;

       CloseHandle (semaphore);
 printf( "Semaphore deleted.\n" );
 printf( "Stop.\n" );

 return 0;
}
```

Listing 26 illustrates the message IPC codes as an example to support the named semaphore shared in the processes.

Listing 26. Equivalent Linux inter-process sempahore process 1 sample code

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>

#define WAIT_FOR_ENTER  printf( "Press ENTER\n" );getchar()

struct msgbuf {
        long mtype;         /* type of message */
        char mtext[1];      /* message text */
};

int main()
{
        key_t           msgKey;
        int             flag;
        struct msgbuf   buff;
        int             sem;
        int             nRet =0;

 printf( "Inter Process Semaphore test - Process 1.\n" );
 printf( "Start.\n" );

 flag = IPC_CREAT|IPC_EXCL;

      if( ( msgKey = (key_t) atol( "456789" ) ) <= 0 )
                return 1;

      flag |= S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;
      sem  = (int) msgget( msgKey, flag );

      if (sem == -1)
             if( errno == EEXIST )
             {
                    flag &= ~IPC_EXCL;
                    sem = (int) msgget( msgKey, flag );
                    if (msgctl(sem, IPC_RMID, NULL ) != 0)
                            return 1;

                    sem = (int) msgget( msgKey, flag );
                    if (sem == -1)
                            return 1;
             }
             else
                    return 1;

      printf( "Semaphore created. \n" );

      WAIT_FOR_ENTER;

      buff.mtype = 123;

      if( msgsnd( sem, &buff, 1, 0 ) < 0 )
          return 1;

      printf( "Semaphore Posted. \n" );

      WAIT_FOR_ENTER;

      if( msgrcv( sem, &buff, 1, 0, 0 ) < 0 )
          return 1;

      printf( "Wait for Semaphore. \n" );
      WAIT_FOR_ENTER;

      msgctl(sem, 0, IPC_RMID );

      printf( "Semaphore deleted.\n" );
 printf( "Stop.\n" );

 return 0;
}
```

Listing 27. Win32 Inter-process semaphore process 2 sample code

```
#include <stdio.h>
#include <windows.h>

int main()
{
 HANDLE semaphore;
 DWORD   retVal;

 printf( "Inter Process Semaphore test - Process 2.\n" );
 printf( "Start.\n" );

 SECURITY_ATTRIBUTES  sec_attr;

       sec_attr.nLength              = sizeof( SECURITY_ATTRIBUTES );
       sec_attr.lpSecurityDescriptor = NULL;
       sec_attr.bInheritHandle       = TRUE;

 semaphore = CreateSemaphore( &sec_attr, 0, 65536, ?456789" );
 if( semaphore == (HANDLE) NULL )
  return RC_OBJECT_NOT_CREATED;

 printf( "Semaphore opened. (%lx)\n", nRet );

 printf( "Try to wait for semaphore.\n" );

       while( ( retVal = WaitForSingleObject( semaphore, 250 ) ) == WAIT_TIMEOUT)
  printf( "Timeout. \n");

       printf( "Semaphore acquired. \n");
       printf( "Try to post the semaphore.\n" );

       if( ! ReleaseSemaphore(semaphore, 1, NULL) )
      return RC_SEM_POST_ERROR;

       printf( "Semaphore posted. \n");

 CloseHandle(semaphore);

 printf( "Semaphore closed. \n");
 printf( "Stop.\n" );

 return 0;
}
```

Listing 28. Equivalent Linux inter-process sempahore process 2 sample code

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <sys/sem.h>
#include <sys/stat.h>
#include <errno.h>
#include <unistd.h>
```

```
#define RC_TIMEOUT = 3

struct msgbuf {
        long mtype;         /* type of message */
        char mtext[1];      /* message text */
};

int main()
{
        key_t           msgKey;
        int             flag=0;
        struct msgbuf   buff;
        int             sem;
        int             nRet =0;


 printf( "Inter Process Semaphore test - Process 2.\n" );
 printf( "Start.\n" );

      if( ( msgKey = (key_t) atol( "456789" ) ) <= 0 )
            return 1;

      flag |= S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP;

      sem = (int) msgget( msgKey, flag );
      if (sem == -1)
            if( errno == EEXIST )
            {
                   flag &= ~IPC_EXCL;
                   sem = (int) msgget( msgKey, flag );
                   if (msgctl(sem, IPC_RMID, NULL ) != 0)
                          return 1;

                   sem = (int) msgget( msgKey, flag );
                   if (sem == -1)
                           return 1;
            }
            else
                   return 1;

 printf( "Semaphore opened. (%lx)\n", nRet );

 if( nRet != 0 )
          return 0;

 printf( "Try to wait for semaphore.\n" );

 while( ( nRet = sem_shared_wait_timed( sem, 250 ) ) == 3)

 printf( "Timeout. (%lx)\n", nRet );
 printf( "Semaphore acquired. (%lx)\n", nRet );
 printf( "Try to post the semaphore.\n" );

       buff.mtype = 123;
       if( msgsnd( sem, &buff, 1, 0 ) < 0 )
          return 1;

 printf( "Semaphore posted. (%lx)\n", nRet );

 if( nRet != 0 )
  return 0;

 printf( "Semaphore closed. (%lx)\n", nRet );
 printf( "Stop.\n" );

 return 0;
}

int sem_shared_wait_timed( int sem, unsigned long timelimit)
{
        struct msgbuf           buff;
        struct timeval          timeOut;
        int                     msg[1];
        int                     nRet=0;

        timeOut.tv_sec  = timelimit / 1000;
        timeOut.tv_usec = (timelimit % 1000) * 1000;

        msg[0] = sem;

        nRet = select( 0x1000, (fd_set *)msg, NULL, NULL, &timeOut );

        if(nRet == 0)
           return 3;

        if( msgrcv( sem, &buff, 1, 0, 0 ) < 0 )
              return 1;
}
```

Conclusion

This third article in a series covered the mapping of Win32 to Linux with respect to semaphore APIs, along with semaphore sample codes for your reference. The threaded, synchronized systems present significant challenges not only in the design and implementation, but also in all stages of quality assurance. Use these articles as a reference when you undertake the migration activity involving Win32 to Linux. Be sure to read the previous articles in this series.

Related topic

- Be sure to read the previous two installments in the series, "Migrate Win32 C/C++ applications to Linux on POWER, Part 1. Process, thread, and shared memory services" (developerWorks, June 2004), and "Migrate Win32 C/C++ applications to Linux on POWER, Part 2. Mutexes" (developerWorks, February 2005).